# Building and using pluggable type systems

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Mahmood Ali
MIT CSAIL
mali@csail.mit.edu

## Abstract

Are you a practitioner who is tired of null pointer exceptions, unintended side effects, SQL injections, concurrency errors, mistaken equality tests, and other run-time errors that appear during testing or in the field? A pluggable type system can guarantee the absence of these errors, and many more.

Are you a researcher who wants to be able to quickly and easily implement a type system, giving you the ability to evaluate it in practice and to field it? You need a framework that supports these essential activities.

This demo is aimed at both audiences. It describes both the theory of pluggable types and also the practice of implementing them. A simple pluggable type-checker can be implemented in 2 minutes, and can be enhanced thereafter. A type checkers is easy to create, easy for programmers to use, and effective in finding real, important bugs.

The demo uses the Checker Framework, which enables you to create pluggable type systems for Java, while your code remains backward-compatible with all versions of Java. The ideas translate to other languages and toolsets. The tools are freely available at `http://types.cs.washington.edu/checker-framework/`.

**Categories and Subject Descriptors**
D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors

**General Terms**
Design, Documentation, Experimentation, Languages, Reliability, Security, Verification

**Keywords**
Pluggable type-system, user-defined type system, Checker Framework, Java, type checking

## 1. Motivation

Buggy software is costly to society: in 2002, insufficient software testing and verification were estimated to cost the US economy $22–60 billion annually [9]. One approach to reducing this cost is program verification via type systems. A static type system helps programmers to detect and prevent errors. However, a language's built-in type system does not help to detect and prevent *enough* errors, because it cannot express certain important invariants. A user-defined, or pluggable, type system enriches the built-in type system by expressing extra information about types via type qualifiers. Example qualifiers include `nonnull`, `readonly`, `interned`, `locked`, and `tainted`, as well as much more complex type systems. Pluggable types permit more expressive compile-time checking and guarantee the absence of additional errors.

We present the Checker Framework for defining pluggable types in a backward-compatible way, in the context of a real-world programming language. Our implementation is for Java, but the ideas and techniques translate to other languages. The Checker Framework is available in source and binary forms from `http://types.cs.washington.edu/checker-framework/`.

The framework makes it easy to define type-checkers either declaratively or procedurally. The framework makes use of a syntax for type qualifiers that is usable today and is planned for inclusion in a future version of the Java language. The framework is in daily academic use by type system designers who are creating new type systems and evaluating them in a realistic context. The framework is in daily industrial use by programmers who want a guarantee that certain types of errors cannot occur. The framework has been applied to millions of lines of code and has found errors in every codebase to which it has been applied.

The framework ships with the following checkers:

- Nullness checker for null pointer errors
- Interning checker for errors in equality testing and interning
- IGJ checker for mutation errors (incorrect side effects), based on the IGJ type system
- Javari checker for mutation errors (incorrect side effects), based on the Javari type system
- Lock checker for concurrency and lock errors, inspired by the Java Concurrency in Practice (JCIP) [5] annotations
- Fake enum checker for integers used as enumerations
- Tainting checker for trust and security errors
- Linear checker to control aliasing and prevent re-use
- Regex checker to prevent use of syntactically invalid regular expressions
- Internationalization checker to ensure that code is properly internationalized: user-visible text is obtained from a localization resource, and proper keys are used for a localization resource

*FSE-18,* November 7–11, 2010, Santa Fe, New Mexico, USA.
ACM 978-1-60558-791-2/10/11.

- Property file checker to check keys used for property files and resource bundles, and to check internationalization and compiler messages
- Basic checker for customized checking without writing any code

Additional checkers written by third parties are also available. The Checker Framework comes with a 100-page manual, mostly describing each of the pluggable type systems.

## 2. Benefits to the profession

Program types are the shining success of formal methods. Types are widely adopted by rank-and-file programmers to detect errors and — more importantly — to verify that no more errors (of particular varieties) exist. However, the uptake of types into practice has been limited by their migration into mainstream programming languages, a process that takes decades. Our work shortcuts this process, offering benefits both to practitioners and to researchers.

For practitioners, we offer the ability to adopt new type systems without breaking compatibility with existing programs, systems, and languages. This enables faster adoption of new ideas. Our work also permits programmers to define application-specific type systems that verify important properties of their systems. We believe that these changes have the potential to transform the way that software is written, and the way that programmers think about their programs. Rather than testing to try to eliminate as many bugs as possible, types provide a pathway to verification and a mindset of writing correct code.

For researchers, we offer the opportunity to experiment with type systems in the context of a widely-used industrial language, Java. To date, evaluating a new type system has generally required either defining a new language, or extensive and incompatible changes to an existing language. Incompatibility with existing tools and programs limits the scope of experimentation, and it limits adoption in practice. This has too often led to flawed theory — say, the approach is unscalable, or it does not handle features like iterators, generics, or the visitor pattern, or a proof of soundness makes unrealistic simplifications. Empirical evaluation is not optional — it is a necessary part of programming language research. By lowering the bar to experimentation, while retaining compatibility with existing tools and programs, we hope to change the way that researchers think about implementing and evaluating their ideas. The result should be more relevant and worthwhile theory and systems, achieved more quickly.

## 3. Related work

This document only skims the surface of related work. For a more extensive discussion, please see [8].

The most closely related frameworks are JQual [6], and JavaCOP [1]. These two frameworks, like the Checker Framework, have been used to implement the Javari [10] type system for enforcing reference immutability. The version implemented in our framework supports the entire Javari language (5 keywords). The JQual and JavaCOP versions have only partial support for 1 keyword (`readonly`), and neither one properly implements method overriding, a key feature of an object-oriented language. Neither JQual nor JavaCOP scales to real programs — either in terms of program size or of language features.

The JastAdd extensible Java compiler [2] includes a module for checking and inferencing of non-null types [3], though it is less featureful and correct than our nullness checker. JastAdd could theoretically be used as a framework to build other type systems.

The goal of a bug-finder such as FindBugs [7] is to find just a few errors, not all errors. FindBugs discards most reports to avoid false positives. Its benefit is that it requires few user annotations.

The Type Annotations (JSR 308) Specification [4] explains how Java will support expressing pluggable type systems.

## References

[1] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 57–74, Portland, OR, USA, October 24–26, 2006.

[2] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 1–18, Montréal, Canada, October 23–25, 2007.

[3] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, October 2007.

[4] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[5] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[6] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 321–336, Montréal, Canada, October 23–25, 2007.

[7] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, Lisbon, Portugal, September 5–6, 2005.

[8] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[9] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, National Institute of Standards and Technology, May 2002.

[10] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.